# Inferring Safe Maude Programs with ÁTAME*

M. Alpuente[1], D. Ballis[2], and J. Sapiña[1]

[1] DSIC-ELP, Universitat Politècnica de València
Camino de Vera s/n, Apdo 22012, 46071 Valencia, Spain
{alpuente,jsapina}@dsic.upv.es
[2] DMIF, University of Udine,
Via delle Scienze, 206, 33100, Udine, Italy
demis.ballis@uniud.it

**Abstract.** In this paper, we present ÁTAME, an assertion-based program specialization tool for the multi-paradigm language Maude. The program specializer ÁTAME takes as input a set $\mathcal{A}$ of system assertions that model the expected program behavior plus a Maude program $\mathcal{R}$ to be specialized that might violate some of the assertions in $\mathcal{A}$. The outcome of the tool is a safe program refinement $\mathcal{R}'$ of $\mathcal{R}$ in which every computation is a good run, i.e., it satisfies the assertions in $\mathcal{A}$. The specialization technique encoded in ÁTAME is fully automatic and ensures that no good run of $\mathcal{R}$ is removed from $\mathcal{R}'$, while the number of bad runs is reduced to zero. We demonstrate the tool capabilities by specializing an overly general nondeterministic dam controller to fulfill a safety policy given by a set of system assertions.

**Keywords:** Program specialization, program adaptability, assertions, Maude, rewriting logic

## 1 Introduction

Adaptability refers to the ability of a piece of software to satisfy requirements dedicated to the specific context in which it is used. In concurrent object-oriented software, adaptability is very fragile as the slightest attempt to modify the foundation of any program component may damage the whole system, ruining the effectiveness of standard reusing mechanisms.

Maude is a high-level programming language and system that supports functional, concurrent, logic, and object-oriented computations and provides equational reasoning modulo algebraic axioms such as associativity, commutativity, and identity. In this paper, we propose an adaptation technique for Maude programs that integrates system assertions and program specialization.

In the literature, program specialization is often used to mean partial evaluation [5], which takes a program of $n$ inputs and produces a simpler and usually

*This work has been partially supported by the EU (FEDER) and the Spanish MINECO under grants TIN2015-69175-C4-1-R, and by Generalitat Valenciana ref. PROMETEOII/2015/013.

faster version where some of the inputs are fixed to particular values. In this paper, we consider a somehow dual specialization transformation where we take a program of $n$ outputs, or more generally, a program that explores $n$ execution traces, and then we produce a more specific version of the original program where we disregard some of the output traces according to the assertional constraints being considered.

Our specialization technique works with Maude programs that are equipped with system assertions, with each assertion consisting of a pair $\Pi \mid \varphi$ where $\Pi$ (the *state template*) is a term and $\varphi$ (the *state invariant*) is a quantifier-free first-order formula with equality that defines a safety property $\varphi$ which must be enforced on all the system states that match (modulo equations and axioms) the state template $\Pi$. In our technique, assertions take an active role since they are directly embedded into the specialized program to safely guide its execution. Given a set of system assertions $\mathcal{A}$ and an overly general Maude program $\mathcal{R} = (\Sigma, E, R)$ (i.e., a program that deploys all desired traces but may disprove some of the assertions), our transformation coerces $\mathcal{R}$ into a specialized program $\mathcal{R}'$ that enforces $\mathcal{A}$. This means that: (i) every execution of $\mathcal{R}'$ is an execution of $\mathcal{R}$ (i.e., no spurious computation states are produced); and (ii) every assertion in $\mathcal{A}$ is satisfied by all computation states in $\mathcal{R}'$. The program $\mathcal{R}'$ is obtained from $\mathcal{R}$ by inserting suitable conditions (abetted by the assertions of $\mathcal{A}$) in the rules of $R$ and defining them by means of new equations that are added to $E$ until a suitable adaptation of the original program is automatically inferred which satisfies all the assertions.

The advantage of this technique is that more refined versions of a program can be incrementally built without any programming effort by simply adding new logical constraints into the given assertion set. Specifically, this makes it possible to adapt existing Maude programs to predefined safety policies and allows the inexperienced user to largely forget about Maude syntax and semantics.

This paper is organized as follows. After some technical preliminaries in Section 2, we introduce a running example that we use to illustrate the kind of specialization that we aim to produce automatically. Section 3 shows how safety policies can actually be defined as system assertions in our rewriting setting, and then applied for program specialization. Section 4 shows how software adaptation can be performed efficiently in the ÁTAME system, which implements our specialization methodology. Section 5 concludes the paper.

## 2 Modeling Software Systems in Maude

Nondeterministic as well as concurrent software systems can be formalized through Maude programs. A Maude program essentially consists of two components, $E$ and $R$, where $E$ is a canonical (membership) equational theory that models system states as terms of an algebraic data type, and $R$ is a set of rewrite rules that define transitions between states. Algebraic structures often involve axioms like associativity (`assoc`), commutativity (`comm`), and/or identity (also known as unity) (`id`) of function symbols, which cannot be handled by ordinary

term rewriting but instead are handled implicitly by working with congruence classes of terms. More precisely, the membership equational theory $E$ is decomposed into a disjoint union $E = \Delta \uplus Ax$, where the set $\Delta$ consists of (conditional) equations and membership axioms (i.e., axioms that assert the type or *sort* of some terms) that are implicitly oriented from left to right as rewrite rules (and operationally used as simplification rules), and $Ax$ is a set of algebraic axioms that are implicitly expressed as function attributes and are only used for $Ax$-matching.

The system evolves by rewriting states using *equational rewriting*, i.e., rewriting with the rewrite rules in $R$ *modulo* the equations and axioms in $E$ [7]. Formally, system computations (also called execution traces) correspond to rewrite sequences $t_0 \xrightarrow{r_0}_E t_1 \xrightarrow{r_1}_E \ldots$, where $t \xrightarrow{r}_E t'$ denotes a transition (modulo $E$) from state $t$ to $t'$ via the rewrite rule of $R$ that is uniquely labeled with label $r$. The transition space of all computations in $\mathcal{R}$ from the initial state $t_0$ can be represented as a *computation tree* whose branches specify all of the system computations in $\mathcal{R}$ that originate from $t_0$.

The following Maude program will be used as a running example throughout the paper.

*Example 1.* Consider a Maude program $\mathcal{R}_{\texttt{DAM}}$ that models a simplified, nondeterministic dam controlling system to monitor and manage the water volume of a given basin[1]. In the program code, variable names are fully capitalized.

We assume that the dam is provided with three spillways called `s1`, `s2`, and `s3` each of which has 4 possible aperture widths of increasing discharge capacity `close`, `open1`, `open2`, `open3`. Each spillway is formally specified by a term `[S,O]`, where `S` $\in \{$`s1`, `s2`, `s3`$\}$ and `O` $\in \{$`close`, `open1`, `open2`, `open3`$\}$. A global spillway configuration is a multiset `[s1,O1] [s2,O2] [s3,O3]` that groups together the three spillways by means of the usual associative and commutative infix, union operator `__` (written in mixfix notation with empty syntax) whose identity is the constant `empty`. System states are defined by terms of the form `{ SC | V | T | AC }` where `SC` is a global spillway configuration, `V` is a rational number that indicates the basin water volume (in $m^3$), `T` is a natural number that timestamps the current configuration, and the Boolean flag `AC`, called `apertureCommand`, enables changes of the spillway aperture widths only when its value is `true`.

Figure 1 shows the equational specification that formalizes basin water inflow and outflow. To keep the exposition simple, we assume that the basin water inflow is constant, while the basin outflow depends on the width of the spillway apertures and can be computed as the sum of the outflows of each spillway in the spillway configuration. Note that inflow and outflow values are measured in $m^3/min$ and are hard-coded into the dam controller. More realistic scenarios could be easily defined by sophisticating the basin inflow and outflow functions.

[1] Maude's syntax is hopefully self-explanatory. Due to space limitations and for the sake of clarity, we only highlight those details of the system that are relevant to this work. A complete Maude specification of the dam controller is available at the ÁTAME website at `http://safe-tools.dsic.upv.es/atame`. For more information about the Maude language, see [4].

```
eq inflow = 3000 .                --- Basin water inflow
eq aperture(close) = 0 .          --- Outflow for a closed spillway
eq aperture(open1) = 200 .        --- Outflow for aperture width open1
eq aperture(open2) = 400 .        --- Outflow for aperture width open2
eq aperture(open3) = 1200 .       --- Outflow for aperture width open3

--- Basin water outflow for a given spillway configuration
eq outflow(empty) = 0 .
eq outflow([S,O] SS) = aperture(O) + outflow(SS) .
```

Fig. 1: Equational definition of basin inflow and outflow.

```
rl [nocmd] : { SC | V | T | true } => { SC | V | T | false } .
rl [openC-1] :
    { [S,close] SS | V | T | true } => { [S,open1] SS | V | T | false } .
rl [open1-2] :
    { [S,open1] SS | V | T | true } => { [S,open2] SS | V | T | false } .
rl [open2-3] :
    { [S,open2] SS | V | T | true } => { [S,open3] SS | V | T | false } .
rl [close1-C] :
    { [S,open1] SS | V | T | true } => { [S,close] SS | V | T | false } .
rl [close2-1] :
    { [S,open2] SS | V | T | true } => { [S,open1] SS | V | T | false } .
rl [close3-2] :
    { [S,open3] SS | V | T | true } => { [S,open2] SS | V | T | false } .
crl [volume] : { SC | V | T | false } => { SC | V' | (T + deltaT) | true }
    if V' := (V + inflow * deltaT) - (outflow(SC) * deltaT) .
```

Fig. 2: (Conditional) rewrite rules for the dam controlling system.

The system dynamics is specified by the eight rewrite rules in Figure 2, which implement system state transitions. The `openX-Y` rewrite rules progressively increment the aperture width of a given spillway (e.g., the rule `open1-2` increases the aperture of the spillway `S` from level `open1` to level `open2`). Dually, `closeX-Y` rewrite rules progressively decrement the aperture width of a spillway. The rule `nocmd` specifies the empty command which basically states that no action is taken on the spillway configuration by the dam controller at time instant `T`. The rule is fired only when the `AC` flag is enabled, and its application disables the flag to allow a new basin water volume to be computed in the next time instant. These eight rules, called *aperture command* rules, implement instantaneous spillway modifications that do not change the time instant or the basin water volume.

The temporal evolution of the basin water volume is specified by the conditional rewrite rule `volume` that computes the volume `V'` at time `T + deltaT`, given the input volume `V` at time `T`. The parameter `deltaT` is measured in minutes and can be set by the user. The volume computation changes the input volume `V` by adding the water inflow and subtracting the corresponding water outflow over the `deltaT` interval.

The use of the `apertureCommand` flag in the rule definitions guarantees a fair interleaving between the applications of the rule `volume` and the remaining aperture command rules. Specifically, this implies that a new basin water volume is computed after each spillway aperture width modification.

Note that computations in $\mathcal{R}_{\texttt{DAM}}$ may reach potentially hazardous system states (e.g., an extremely high water volume). This is because $\mathcal{R}_{\texttt{DAM}}$ does not implement any spillway management policy that safely restricts the applications of the aperture command rules.

## 3 Defining Safety Policies through Assertions

A *safety policy* for a Maude program $\mathcal{R}$ is defined by means of a set $\mathcal{A}$ of system assertions, each assertion being of the form $\Pi \mid \varphi$, which $\mathcal{R}$ must satisfy. Intuitively, system assertions specify those computation states such that, for every subterm of a state that matches the algebraic structure of the state template $\Pi$ with substitution (modulo the axioms) $\sigma$, the constraints given by the instantiated invariant $\varphi\sigma$ are satisfied. Besides the usual Boolean operators and Maude predefined predicates, the state invariant $\varphi$ may include user-defined predicates as well as functions that can be specified via suitable equational definitions.

*Example 2.* Let us consider the user-defined function `openSpillways(SC)` that returns the number of open spillways in the spillway configuration `SC`, whose equational definition is

```
eq openSpillways(empty)    = 0 .
eq openSpillways([S,O] SC) = if (O =/= close)
                               then (1 + openSpillways(SC))
                               else openSpillways(SC)
                               fi .
```

and the safety policy $\mathcal{A}_{\texttt{DAM}}$ of Figure 3 for the dam controller of Example 1 that specifies some safety constraints to prevent basin critical situations.

```
(a1) { SC | V | T | AC } | (V < 50000000)
(a2) { [ S1,O1 ] [ S2,O2 ] [ S3,O3 ] | V:Rat | T:TimeStamp | AC:Bool } |
                (V:Rat > 40000000) implies (
                    (O1 == open3 and O2 =/= close and O3 =/= close) or
                    (O2 == open3 and O1 =/= close and O3 =/= close) or
                    (O3 == open3 and O1 =/= close and O2 =/= close))
(a3) { SC | V | T | AC } | (V < 10000000) implies
                                (openSpillways(SC) == 0)
(a4) { SC | V | T | AC } | ((V >= 10000000) and (V <= 40000000)) implies
                                (openSpillways(SC) == 2)
```

Fig. 3: Safety policy $\mathcal{A}_{\texttt{DAM}}$ for the dam controller $\mathcal{R}_{\texttt{DAM}}$.

More specifically, assertion `a1` states that, in every system state, the basin water volume must be less than 50 million $m^3$ to avoid dam bursts and potentially disastrous floods. Assertion `a2` specifies that, whenever the basin water volume is greater than 40 million $m^3$, all of the spillways must be open and the aperture width of at least one spillway must be maximal (level `open3`). Assertion `a3` requires the closure of all the spillways when the basin water volume is particularly low (10 million $m^3$). Finally, assertion `a4` specifies the spillway handling for an intermediate water volume (10 million $m^3 \leq$ `V` $\leq$ 40 million $m^3$); in this scenario we require that exactly two spillways be constantly open.

## 4 Computing Safe Maude Programs with ÁTAME

Program specialization techniques make it possible to automatically transform a program into a specialized version, according to an execution context. In our approach, we use assertions to set the specialization scenario and guide a two-phase program specialization technique that allows a Maude program $\mathcal{R}$ to be refined into a program $\mathcal{R}'$ w.r.t. a safety policy $\mathcal{A}$ as follows.

The first phase translates the safety policy $\mathcal{A}$ to be fulfilled into an executable equational definition $Eq(\mathcal{A})$ that can be used to detect assertion violations within system states. Roughly speaking, given a system state $t$, a violation of some assertion in $\mathcal{A}$ is detected in $t$ if $t$ can be simplified into the special constant `fail` by using the equational theory $E$ of $\mathcal{R}$ extended with $Eq(\mathcal{A})$.

The second phase transforms the original rewrite rules of $\mathcal{R}$ into guarded, conditional rewrite rules that can only be fired if no system assertion is violated. Intuitively, this is achieved by transforming each rewrite rule $r : (\lambda \Rightarrow \rho$ `if C`$)$ of $\mathcal{R}$ into a refined version $r' : (\lambda \Rightarrow \rho$ `if C` $\wedge$ `check`$(\rho)$ `=/= fail`$)$ of $r$ that contains the extra constraint `check`$(\rho)$ `=/= fail` that holds when (the instances of) the right-hand side $\rho$ cannot by simplified to `fail` by using the extended equational theory $E \cup Eq(\mathcal{A})$. This ensures that any state transition $t_1 \xrightarrow{r'}_{E \cup Eq(\mathcal{A})} t_2$, that yields the system state $t_2$ by means of the application of the rule $r'$, is enabled only if $t_2$ is a safe state, that is, a state that does not violate any assertion.

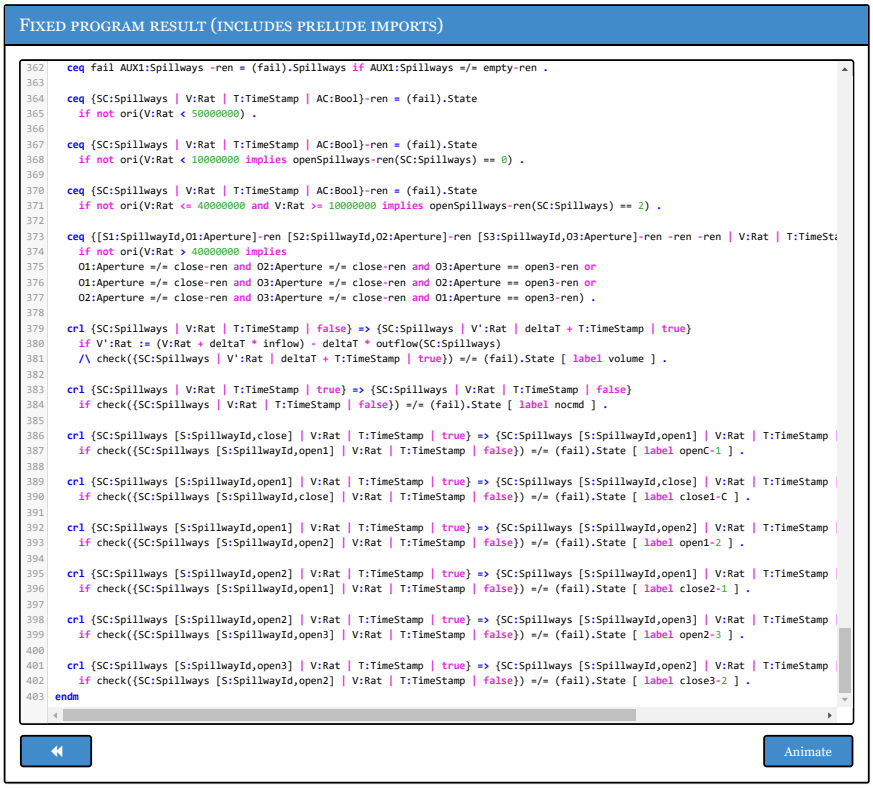
Computations in the resulting program $\mathcal{R}'$ are both reproducible in $\mathcal{R}$ and guaranteed to meet $\mathcal{A}$. In other words, for each computation $\mathcal{C}$ in $\mathcal{R}'$, (i) $\mathcal{C}$ is also a computation in $\mathcal{R}$, and (ii) there is no system state $t$ in $\mathcal{C}$ that violates one or more system assertions of $\mathcal{A}$.

The proposed specialization technique has been efficiently implemented in a Maude tool called ÁTAME (*Assertion-based Theory Amendment in MaudE*) that has been implemented in Maude itself by using Maude's meta-level capabilities. ÁTAME integrates a RESTful Web service that is written in Java, and an intuitive Web user interface that is based on AJAX technology and is written in HTML5 and Javascript. The implementation contains about 600 lines of Maude source code, 600 lines of C++ code, 750 lines of Java code, and 700 lines of HTML5 and JavaScript code.

As an additional feature, ÁTAME provides the interconnection with the ANIMA Maude stepper [1], which integrates program animation capabilities into the

ÁTAME system. Indeed, we can execute the computed specialization by incrementally building and exploring the computation tree of $\mathcal{R}'$ w.r.t. a given input initial state. The tool ÁTAME is publicly available together with a number of examples at `http://safe-tools.dsic.upv.es/atame`.

In order to demonstrate the tool capabilities, in the following we show the specialization of the dam controller $\mathcal{R}_{\texttt{DAM}}$ w.r.t. the safety policy $\mathcal{A}_{\texttt{DAM}}$ that can be achieved by ÁTAME.




```
362   ceq fail AUX1:Spillways -ren = (fail).Spillways if AUX1:Spillways =/= empty-ren .
363
364   ceq {SC:Spillways | V:Rat | T:TimeStamp | AC:Bool}-ren = (fail).State
365     if not ori(V:Rat < 50000000) .
366
367   ceq {SC:Spillways | V:Rat | T:TimeStamp | AC:Bool}-ren = (fail).State
368     if not ori(V:Rat < 10000000 implies openSpillways-ren(SC:Spillways) == 0) .
369
370   ceq {SC:Spillways | V:Rat | T:TimeStamp | AC:Bool}-ren = (fail).State
371     if not ori(V:Rat <= 40000000 and V:Rat >= 10000000 implies openSpillways-ren(SC:Spillways) == 2) .
372
373   ceq {[S1:SpillwayId,O1:Aperture]-ren [S2:SpillwayId,O2:Aperture]-ren [S3:SpillwayId,O3:Aperture]-ren -ren -ren | V:Rat | T:TimeSt
374     if not ori(V:Rat > 40000000 implies
375     O1:Aperture =/= close-ren and O2:Aperture =/= close-ren and O3:Aperture == open3-ren or
376     O1:Aperture =/= close-ren and O3:Aperture =/= close-ren and O2:Aperture == open3-ren or
377     O2:Aperture =/= close-ren and O3:Aperture =/= close-ren and O1:Aperture == open3-ren) .
378
379   crl {SC:Spillways | V:Rat | T:TimeStamp | false} => {SC:Spillways | V':Rat | deltaT + T:TimeStamp | true}
380     if V':Rat := (V:Rat + deltaT * inflow) - deltaT * outflow(SC:Spillways)
381     /\ check({SC:Spillways | V':Rat | deltaT + T:TimeStamp | true}) =/= (fail).State [ label volume ] .
382
383   crl {SC:Spillways | V:Rat | T:TimeStamp | true} => {SC:Spillways | V:Rat | T:TimeStamp | false}
384     if check({SC:Spillways | V:Rat | T:TimeStamp | false}) =/= (fail).State [ label nocmd ] .
385
386   crl {SC:Spillways [S:SpillwayId,close] | V:Rat | T:TimeStamp | true} => {SC:Spillways [S:SpillwayId,open1] | V:Rat | T:TimeStamp
387     if check({SC:Spillways [S:SpillwayId,open1] | V:Rat | T:TimeStamp | false}) =/= (fail).State [ label openC-1 ] .
388
389   crl {SC:Spillways [S:SpillwayId,open1] | V:Rat | T:TimeStamp | true} => {SC:Spillways [S:SpillwayId,close] | V:Rat | T:TimeStamp
390     if check({SC:Spillways [S:SpillwayId,close] | V:Rat | T:TimeStamp | false}) =/= (fail).State [ label close1-C ] .
391
392   crl {SC:Spillways [S:SpillwayId,open1] | V:Rat | T:TimeStamp | true} => {SC:Spillways [S:SpillwayId,open2] | V:Rat | T:TimeStamp
393     if check({SC:Spillways [S:SpillwayId,open2] | V:Rat | T:TimeStamp | false}) =/= (fail).State [ label open1-2 ] .
394
395   crl {SC:Spillways [S:SpillwayId,open2] | V:Rat | T:TimeStamp | true} => {SC:Spillways [S:SpillwayId,open1] | V:Rat | T:TimeStamp
396     if check({SC:Spillways [S:SpillwayId,open1] | V:Rat | T:TimeStamp | false}) =/= (fail).State [ label close2-1 ] .
397
398   crl {SC:Spillways [S:SpillwayId,open2] | V:Rat | T:TimeStamp | true} => {SC:Spillways [S:SpillwayId,open3] | V:Rat | T:TimeStamp
399     if check({SC:Spillways [S:SpillwayId,open3] | V:Rat | T:TimeStamp | false}) =/= (fail).State [ label open2-3 ] .
400
401   crl {SC:Spillways [S:SpillwayId,open3] | V:Rat | T:TimeStamp | true} => {SC:Spillways [S:SpillwayId,open2] | V:Rat | T:TimeStamp
402     if check({SC:Spillways [S:SpillwayId,open2] | V:Rat | T:TimeStamp | false}) =/= (fail).State [ label close3-2 ] .
403 endm
```

Fig. 4: A fragment of the safe specialization for $\mathcal{R}_{\texttt{DAM}}$ computed by ÁTAME.

*Example 3.* By feeding the ÁTAME system with the Maude program for the dam controller $\mathcal{R}_{\texttt{DAM}}$ of Example 1 and the safety policy $\mathcal{A}_{\texttt{DAM}}$ of Example 2, a program specialization $\mathcal{R}'_{\texttt{DAM}}$ for $\mathcal{R}_{\texttt{DAM}}$ is automatically computed. Figure 4 shows a fragment of such a specialization that includes $Eq(\mathcal{A}_{\texttt{DAM}})$ (i.e., the equations for detecting assertion violations) and the constrained, conditional versions of the original rewrite rules. Note that all the operators in the equations of $Eq(\mathcal{A}_{\texttt{DAM}})$ are renamed by adding the textual suffix `-ren`. This guarantees that assertion checking is orthogonal to system computations, that is, there is no interference between the assertion checking mecha-

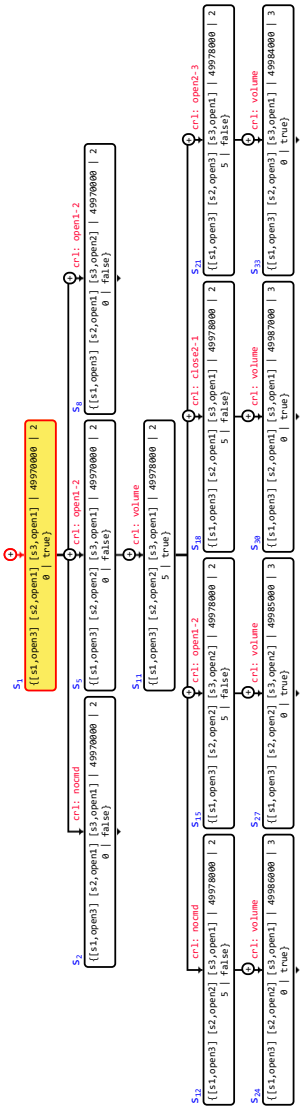


Fig. 5: A computation tree fragment for $\mathcal{R}'_{\texttt{DAM}}$.

nism and the applications of the rewrite rules that make the system evolve only through safe states that meet $\mathcal{A}_{\texttt{DAM}}$. A fragment of the computation tree that is deployed by the Maude stepper ANIMA for the initial state $s$=`{[s1,open3] [s2,open1] [s3,open1] | 49970000 | 20 | true}` in $\mathcal{R}'_{\texttt{DAM}}$ is shown in Figure 5. Note that all of the states in the considered tree fragment fulfill the system assertions formalized in $\mathcal{A}_{\texttt{DAM}}$.

In practice, the runtime cost of checking the assertions must be weighed against the saving gained from embedding them into the code and thus omitting

the need for executing programs within a monitored runtime environment. The manual inclusion of safety policies as a piece of code is generally problematic, since such conditions may not be easily coded by non-specialists. Moreover, as shown in [2], the monitored runtime verification of external constraints generally incurs more cost than running the specialized program that is automatically inferred by our approach. In the case of the running example of this paper, as expected the specialized program $\mathcal{R}'_{\texttt{DAM}}$ is slightly slower than the original program $\mathcal{R}_{\texttt{DAM}}$. Nevertheless, running $\mathcal{R}'_{\texttt{DAM}}$ is 68% faster than running $\mathcal{R}_{\texttt{DAM}}$ within a runtime environment that supports dynamic assertion-checking. As for the time necessary for computing the program specializations, it is almost negligible (a few milliseconds). For a detailed empirical evaluation, we refer to [2].

## 5 Concluding Remarks

The technique described in this paper presents similarities with automated program correction and related problems such as code fixing and repair techniques. The discussion of these similarities is outside the scope of this paper; a detailed comparison can be found in [2]. Loosely related to this work is also the concept of program specialization of terminating programs based on output constraints (i.e., program post-conditions) [6]. This methodology translates the constraints into a characterization function for the program's input that is used to guide a partial evaluation process. In contrast, we deal with non-terminating concurrent programs and the specialization that we achieve cannot be produced by any (conventional or unconventional) partial evaluation techniques for Maude [3]. To our knowledge, the assertion-based functionality for molding programs supported by ÁTAME is beyond the capabilities of all existing Maude tools.

## References


1. Alpuente, M., Ballis, D., Frechina, F., Sapiña, J.: Exploring Conditional Rewriting Logic Computations. Journal of Symbolic Computation **69**, 3–39 (2015)
2. Alpuente, M., Ballis, D., Sapiña, J.: Static Correction of Maude Programs with Assertions. Tech. rep., Universitat Politècnica de València (2018), available at: `http://hdl.handle.net/10251/100268`
3. Alpuente, M., Cuenca-Ortega, A., Escobar, S., Meseguer, J.: Partial Evaluation of Order-sorted Equational Programs modulo Axioms. In: Proceedings of LOPSTR 2016. Lecture Notes in Computer Science, vol. 10184, pp. 3–20. Springer (2016)
4. Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: Maude Manual (Version 2.7.1). Tech. rep., SRI International (2016), available at: `http://maude.cs.uiuc.edu/maude2-manual/`
5. Danvy, O., Glück, R., Thiemann, P. (eds.): Proceedings of the International Seminar on Partial Evaluation (Dagstuhl 1996), Lecture Notes in Computer Science, vol. 1110. Springer (1996)
6. Khoo, S.C., Shi, K.: Program Adaptation via Output-Constraint Specialization. Higher-Order and Symbolic Computation **17**(1), 93–128 (2004)
7. Meseguer, J.: Conditional Rewriting Logic as a Unified Model of Concurrency. Theoretical Computer Science **96**(1), 73–155 (1992)